Austin Neidert
August 15, 2018
OPOS Transaction Mode Tutorial Using MICPosPrinter2 1.14.0.20.pdf

MICPosPrinter2 Transaction Mode Tutorial Using C# Code

This tutorial was created using Visual Studio 2013 and the .NET Framework 4.5. Before this tutorial can be completed POS for .NET 1.14 must be installed. This tutorial assumes the service object has been installed and set up using the using the Microcom OPOS Setup applications. The POS for .NET package should also be referenced in the Visual Studio project. Once the project has a reference to POS for .NET a using directive for the package can be added. At the top of the file in which this code is being entered, add a using directive to reference POS for .NET.

```
using Microsoft.PointOfService;
```

All of the following code (excluding the functions listed on the "supplementary functions" page at the end) should be entered in a function in the order listed here. The first step is to obtain the service object from a list of all installed service objects. To get the list of service objects a PosExplorer object must be created and its GetDevices function should be called. Start by creating a function and adding the following lines to it.

```
PosExplorer explorer = new PosExplorer();
DeviceInfo info = null;
DeviceCollection devices = explorer.GetDevices();
```

Once the list of devices is retrieved the service object needs to be gotten from the list and the PosPrinter object needs to be created.

```
for (int i = 0; i < devices.Count; i++)
{
    if (devices[i].ServiceObjectName == "MICPosPrinter")
    {
        info = devices[i];
        break;
    }
}
PosPrinter _printer = (PosPrinter)explorer.CreateInstance(info);
```

At this point the service object has been instantiated. Before it can be used in the POS system it must be opened, claimed, and enabled.

```
_printer.Open();
_printer.Claim(1000);
_printer.DeviceEnabled = true;
```

Now, to print a transaction, Transaction Mode must be entered.

```
_printer.TransactionPrint(PrinterStation.Receipt, PrinterTransactionControl.Transaction);
```

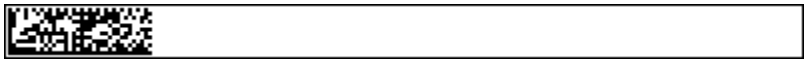
Once in Transaction Mode the transaction can be built. It is necessary to initialize MapMode so that the units being used are known. All POS code is converting to the printer's native language (LDS2).

```
_printer.MapMode = MapMode.English;
```

Printable items may now be added to the transaction. This includes bar codes, ruled lines, bitmaps, and text. First on the list is bar codes. When creating linear bar codes the symbology is specified as the Microsoft.PointOfService.BarCodeSymbology variable (example: BarCodeSymbology.Code128). When specifying the desired width of linear bar codes, the value should be approximated. Bar code width scales by integers so they may not print exactly the desired size. When creating two-dimensional bar codes such as datamatrix and PDF417, the user has the option to either just enter the standard BarCodeSymbology variable (example: BarCodeSymbology.DataMatrix) or there are extra parameters that can be added to the standard variable. Here a function is used to add parameters to a datamatrix symbology variable. Function definitions for adding parameters to datamatrix and pdf417 symbologies will be listed at the end. If the standard bar code symbology variable is specified for a two-dimensional bar code, then auto/default values will be used for the added variables. The generatePDF417Symbology and generateDataMatrixSymbology functions are listed at the end of this document. Note that with two-dimensional bar codes such as datamatrix and PDF417, the height of the bar code is calculated without using the POS parameter. Here 355 is entered to be used as a measurement for how long to extend the page.

```
    string data = "12345678";
    BarCodeSymbology symbology = generateDataMatrixSymbology(12, 36, true, DataMatrixEncodingScheme.Base256);
    _printer.PrintBarCode(PrinterStation.Receipt, data, symbology, 355, 720, 0, BarCodeTextPosition.None);//width
is 720 and height is 0 because with datamatrix only width is used
    _printer.RotatePrint(PrinterStation.Receipt, PrintRotation.Normal | PrintRotation.Bitmap |
PrintRotation.Barcode);
```

The transaction now looks like the following.



Next to add is ruled lines. Ruled lines are drawn with the DrawRuledLine function. This function is only supported after version 1.13 of POS for .Net. Note that in the POS spec. vertical lines are drawn by specifying when to start and when to stop with separate DrawRuledLine calls (so that in line-printing they can be drawn per line). As opposed to the spec., in MICPosPrinter's PageMode functionality they are drawn in the same manner as horizontal lines. This means that drawing vertical lines is done by specifying the start point and the length.

```
    _printer.DrawRuledLine(PrinterStation.Receipt, "0,500", LineDirection.Vertical, 10,
LineStyle.SingleSolidLine, 0);
    _printer.DrawRuledLine(PrinterStation.Receipt, "0,500", LineDirection.Horizontal, 10,
LineStyle.SingleSolidLine, 0);
```

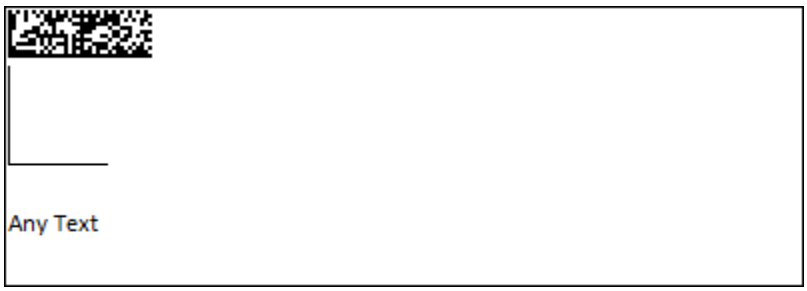
The transaction now looks like the following.

After those are complete all that is left are bitmaps and text elements. When printing text; some attributes of the print to consider are font, alignment, reverse video, and rotation. Below are examples

of how to set some of these. Alignment, reverse video, and font are all set using an escape sequence within PrintNormal. Rotation is set by the RotatePrint function. Note that the setFont function being used here is listed at the end of this document. It uses the standard PrintNormal POS function. If the printer has a font that does not appear in FontTypefaceList, FontTypefaceList can be refreshed (synced with the printer) using DirectIO command number 1. Adding "\n" to a string adds a multiple of RecLineSpacing to how much the page increases in size. The page can be increased in size by N MapMode units by setting RecLineSpacing = 1 and calling PrintNormal(PrinterStation.Receipt, "\n").

```
setFont("@bold_08", _printer);
_printer.PrintNormal(PrinterStation.Receipt, (char)27 + "|lA"); //set alignment to left
_printer.PrintNormal(PrinterStation.Receipt, "\n\nAny Text\n\n");
```

The transaction now looks like the following.



Here is an example of printing a bitmap. SetBitmap is important in a situation where an image has yet to be saved to the printer. SetBitmap removes all ocurrences of ".bmp" and "." in the file name before saving to the printer. If the name without ".bmp" and "." is greater than six characters long, the name becomes the first six characters in the name. In this case "testimage.bmp" would become "testim". The name specified in PrintBitmap must be the name of the image on the printer. The string specified in SetBitmap must include the path and file name of the image on the host computer where in this case "c:\\" is the path and "test.bmp" is the file name. A function to convert from the host computer's file path and file name to the printer's file name is listed at the end. This function is named "getNameOnPrinter". After the first time SetBitmap is called it can be removed because the test bitmap will be saved on the printer. Note that in RotatePrint using " | PrintRotation.Bitmap" is only used here to be explicit. Ordinarily it is only used when a rotation angle other than 0 is specified.

```
    string imagePathAndName = "c:\\test.bmp";
    _printer.SetBitmap(0, PrinterStation.Receipt, imagePathAndName, PosPrinter.PrinterBitmapAsIs,
PosPrinter.PrinterBitmapLeft);
    _printer.RotatePrint(PrinterStation.Receipt, PrintRotation.Normal | PrintRotation.Bitmap);
    string nameOnPrinter = getNameOnPrinter(imagePathAndName);
    _printer.PrintBitmap(PrinterStation.Receipt, nameOnPrinter, PosPrinter.PrinterBitmapAsIs,
PosPrinter.PrinterBitmapLeft);
```

Depending on the image used the transaction will now look like the following.

At this point the transaction has been completely built. The TransactionPrint function is used to cause the transaction to print and clear.

```
_printer.TransactionPrint(PrinterStation.Receipt, PrinterTransactionControl.Normal);
```

When use of the printer has concluded it is important to close it. Closing the printer will stop extra threads that the service object starts as well as allow the printer to be used by another application.

```
_printer.Close();
```

Here is the code for the entire tutorial.

```
    PosExplorer explorer = new PosExplorer();
    DeviceInfo info = null;
    DeviceCollection devices = explorer.GetDevices();
    for (int i = 0; i < devices.Count; i++)
    {
        if (devices[i].ServiceObjectName == "MICPosPrinter")
        {
            info = devices[i];
            break;
        }
    }
    PosPrinter _printer = (PosPrinter)explorer.CreateInstance(info);
    _printer.Open();
    _printer.Claim(1000);
    _printer.DeviceEnabled = true;
    _printer.TransactionPrint(PrinterStation.Receipt, PrinterTransactionControl.Transaction);
    _printer.MapMode = MapMode.English;
    string data = "12345678";
    BarCodeSymbology symbology = generateDataMatrixSymbology(12, 36, true, DataMatrixEncodingScheme.Base256);
    _printer.PrintBarCode(PrinterStation.Receipt, data, symbology, 355, 720, 0, BarCodeTextPosition.None);//width
is 720 and height is 0 because with datamatrix only width is used
    _printer.RotatePrint(PrinterStation.Receipt, PrintRotation.Normal | PrintRotation.Bitmap |
PrintRotation.Barcode);
    _printer.DrawRuledLine(PrinterStation.Receipt, "0,500", LineDirection.Vertical, 10,
LineStyle.SingleSolidLine, 0);
    _printer.DrawRuledLine(PrinterStation.Receipt, "0,500", LineDirection.Horizontal, 10,
LineStyle.SingleSolidLine, 0);
    setFont("@bold_08", _printer);
    _printer.PrintNormal(PrinterStation.Receipt, (char)27 + "|lA"); //set alignment to left
    _printer.PrintNormal(PrinterStation.Receipt, "\n\nAny Text\n\n");
    string imagePathAndName = "c:\\test.bmp";
    _printer.SetBitmap(0, PrinterStation.Receipt, imagePathAndName, PosPrinter.PrinterBitmapAsIs,
PosPrinter.PrinterBitmapLeft);
    _printer.RotatePrint(PrinterStation.Receipt, PrintRotation.Normal | PrintRotation.Bitmap);
    string nameOnPrinter = getNameOnPrinter(imagePathAndName);
    _printer.PrintBitmap(PrinterStation.Receipt, nameOnPrinter, PosPrinter.PrinterBitmapAsIs,
PosPrinter.PrinterBitmapLeft);
    _printer.TransactionPrint(PrinterStation.Receipt, PrinterTransactionControl.Normal);
    _printer.Close();
```

This code can be found in the Tutorial Application Visual Studio project, in the Form1.cs file, inside the `TransactionModeTutorialButton_Click(object sender, EventArgs e)` function. Note that this code has been slightly modified so that the printer is not closed and is not opened multiple times.

Supplementary Functions

Below is the function that is used to select the font from the FontTypefaceList. When sending the escape sequence for font selection, the POS spec. says that the number specified is the number of font in the list (1 based). This means that the first font in the list is number 1. In C# arrays like FontTypefaceList are 0 based. For that reason, (i + 1) is used to determine fontNumber.

```
    private void setFont(string fontName, PosPrinter printer)
    {
        for (int i = 0; i < printer.FontTypefaceList.Length; i++)
        {
            if (printer.FontTypefaceList[i] == fontName)
            {
                string fontNumber = (i + 1).ToString();
                printer.PrintNormal(PrinterStation.Receipt, (char)27 + "|" + fontNumber + "fT");
                break;
            }
        }
    }
```

Below is the function that is used to add parameters to the datamatrix symbology along with an enum used as one of those parameters.

```
    enum DataMatrixEncodingScheme { auto = 0, ASCII = 1, C40 = 2, Text = 3, Base256 = 4 }
    private BarCodeSymbology generateDataMatrixSymbology(int leftDimension, int rightDimension, bool
tildeIsControlCharacter, DataMatrixEncodingScheme scheme)
    {
        BarCodeSymbology symbology = BarCodeSymbology.DataMatrix;
        symbology |= (BarCodeSymbology)(leftDimension << 24);
        symbology |= (BarCodeSymbology)(rightDimension << 16);
        symbology |= (BarCodeSymbology)(((int)scheme) << 13);
        if (tildeIsControlCharacter)
            symbology |= (BarCodeSymbology)(1 << 12);
        return symbology;
    }
```

Below is the function that is used to add parameters to the pdf417 symbology.

```
     private BarCodeSymbology generatePDF417Symbology(int scaleVertically, int scaleHorizontally, int codewords,
int ECCNumber)
    {
        BarCodeSymbology symbology = BarCodeSymbology.Pdf417;
        symbology |= (BarCodeSymbology)(scaleVertically << 25);
        symbology |= (BarCodeSymbology)(scaleHorizontally << 18);
        symbology |= (BarCodeSymbology)(codewords << 13);
        symbology |= (BarCodeSymbology)(ECCNumber << 9);
        return symbology;
    }
```

Below is the function “getNameOnPrinter”. This function converts from the host computer’s file path and file name to the file name that is on the printer. For example, it converts “c:\\test.bmp” to “test”.

```
    public string getNameOnPrinter(string pathAndName)
    {
        string nameOnPrinter = pathAndName.Substring(pathAndName.LastIndexOf("\\") + 1).Replace(".bmp",
"").Replace(".", "");
        if (nameOnPrinter.Length > 6)
            nameOnPrinter = nameOnPrinter.Substring(0, 6);
        return nameOnPrinter;
    }
```